Chapter 3
Agile Principles

Before we delve deeper into the mechanics of Scrum, it will be helpful to understand the underlying principles that drive and inform those mechanics.

This chapter describes the agile principles that underlie Scrum and compares them with those of traditional, plan-driven, sequential product development. In doing so, the chapter sets the stage for understanding how Scrum differs from more traditional forms of product development and for a more detailed analysis of Scrum practices in subsequent chapters.

Overview

I find it instructive to introduce Scrum's underlying principles by comparing them with the beliefs that drive more traditional, plan-driven, sequential development. Doing so makes it easier for people to understand how Scrum is similar to or different from something they know and understand.

The goal of comparing agile principles with traditional development principles is not to make the case that plan-driven, sequential development is bad and that Scrum is good. Both are tools in the professional developer's toolkit; there is no such thing as a bad tool, rather just inappropriate times to use that tool. As I described briefly in the context of the Cynefin framework in Chapter 1, Scrum and traditional, plan-driven, sequential development are appropriate to use on different classes of problems.

In making the comparison between the two approaches, I am using the pure or "textbook" description of plan-driven, sequential development. By taking this perspective when describing traditional development, I am better able to draw out the distinctions and more clearly illustrate the principles that underlie Scrum-based development.

One pure form of traditional, plan-driven development frequently goes by the term **waterfall** (see Figure 3.1). However, that is just one example of a broader class of **plan-driven** processes (also known as **traditional**, **sequential**, **anticipatory**, **predictive**, or **prescriptive development processes**).

Plan-driven processes are so named because they attempt to plan for and anticipate up front all of the features a user might want in the end product, and to determine how best to build those features. The idea here is that the better the planning, the better the understanding, and therefore the better the execution. Plan-driven processes are often called sequential processes because practitioners perform, in sequence, a complete requirements analysis followed by a complete design followed in turn by coding/building and then testing.

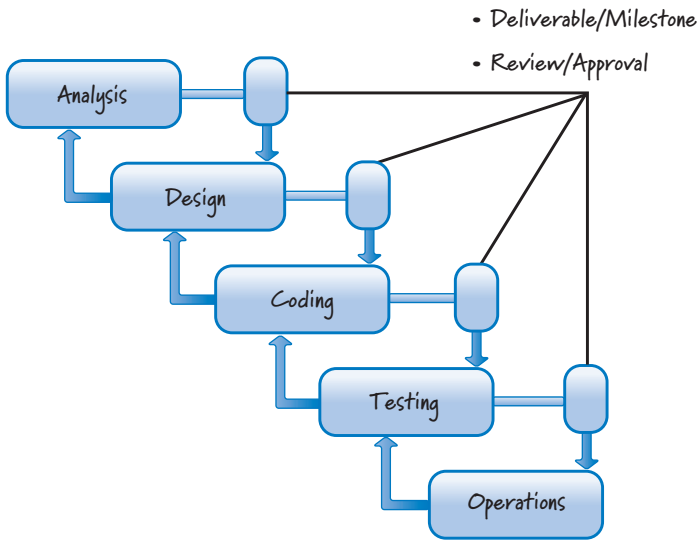

FIGURE 3.1 Waterfall process

Plan-driven development works well if you are applying it to problems that are well defined, predictable, and unlikely to undergo any significant change. The problem is that most product development efforts are anything but predictable, especially at the beginning. So, while a plan-driven process gives the impression of an orderly, accountable, and measurable approach, that impression can lead to a false sense of security. After all, developing a product rarely goes as planned.

For many, a plan-driven, sequential process just makes sense, understand it, design it, code it, test it, and deploy it, all according to a well-defined, prescribed plan. There is a belief that it *should* work. If applying a plan-driven approach doesn't work, the prevailing attitude is that we must have done something wrong. Even if a plan-driven process repeatedly produces disappointing results, many organizations continue to apply the same approach, sure that if they just do it better, their results will improve. The problem, however, is not with the execution. It's that plan-driven approaches are based on a set of beliefs that do not match the uncertainty inherent in most product development efforts.

Scrum, on the other hand, is based on a different set of beliefs—ones that do map well to problems with enough uncertainty to make high levels of predictability difficult. The principles that I describe in this chapter are drawn from a number of sources, including the Agile Manifesto (Beck et al. 2001), lean product development (Reinertsten 2009b; Poppendieck and Poppendieck 2003), and "The Scrum Guide" (Schwaber and Sutherland 2011).

These principles are organized into several categories as shown in Figure 3.2.

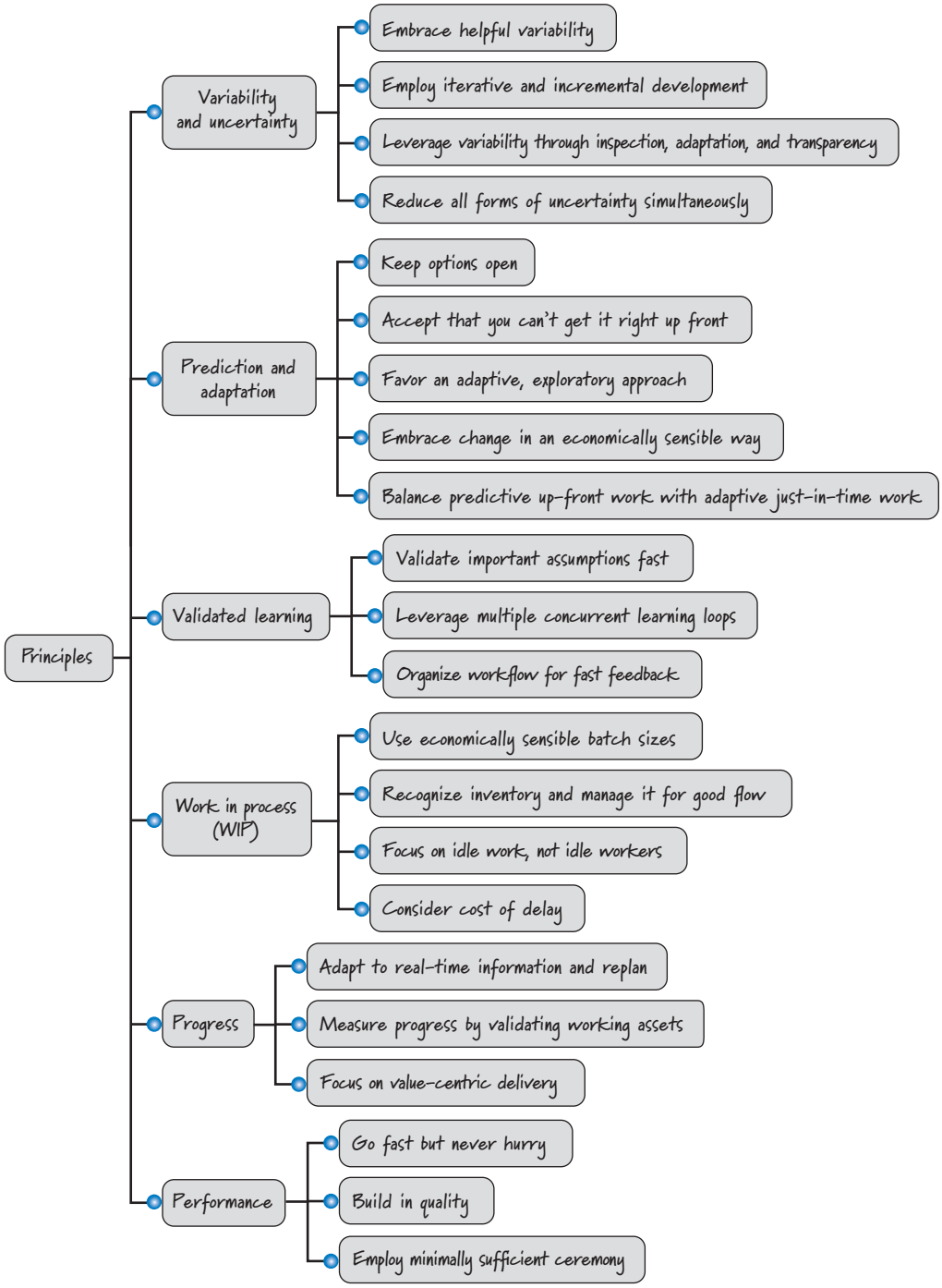

FIGURE 3.2 Categorization of principles

I start by discussing principles that leverage the variability and uncertainty inherent in product development. This is followed by a discussion of principles that deal with balancing up-front prediction with just-in-time adaptation. Then, I discuss principles focused on learning, followed by principles for managing the work in process. I conclude by focusing on progress and performance principles.

Variability and Uncertainty

Scrum leverages the **variability** and **uncertainty** in product development to create innovative solutions. I describe four principles related to this topic:

- Embrace helpful variability.
- Employ iterative and incremental development.
- Leverage variability through inspection, adaptation, and transparency.
- Reduce all forms of uncertainty simultaneously.

Embrace Helpful Variability

Plan-driven processes treat product development like manufacturing—they shun variability and encourage conformance to a **defined process**. The problem is that product development is not at all like product manufacturing. In manufacturing our goal is to take a fixed set of requirements and follow a sequential set of well-understood steps to manufacture a finished product that is the same (within a defined variance range) every time (see Figure 3.3).

In product development, however, the goal is to create the unique *single instance* of the product, not to *manufacture* the product. This single instance is analogous to a unique recipe. We don't want to create the same recipe twice; if we do, we have wasted our money. Instead, we want to create a unique recipe for a new product. Some amount of variability is necessary to produce a different product each time. In fact, every feature we build within a product is different from every other feature within that product, so we need variability even at this level. Only once we have the recipe do we manufacture the product—in the case of software products, as easily as copying bits.

That being said, some manufacturing concepts do apply to product development and can and should be leveraged. For example, as I will discuss shortly, recognizing and managing **inventory** (or work in process), which is essential to manufacturing, is

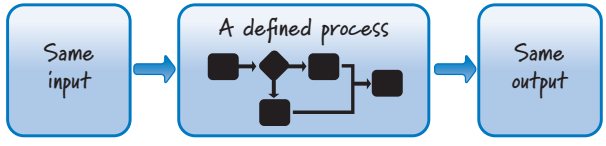

FIGURE 3.3 Defined process

also essential in product development. By the very nature of the work involved, however, product development and product manufacturing are not at all the same thing and as such require very different processes.

Employ Iterative and Incremental Development

Plan-driven, sequential development assumes that we will get things right up front and that most or all of the product pieces will come together late in the effort.

Scrum, on the other hand, is based on **iterative and incremental development**. Although these two terms are frequently used as if they were a single concept, iterative development is actually distinct from incremental development.

Iterative development acknowledges that we will probably get things wrong before we get them right and that we will do things poorly before we do them well (Goldberg and Rubin 1995). As such, iterative development is a planned rework strategy. We use multiple passes to improve what we are building so that we can converge on a good solution. For example, we might start by creating a prototype to acquire important knowledge about a poorly known piece of the product. Then we might create a revised version that is somewhat better, which might in turn be followed by a pretty good version. In the course of writing this book, for example, I wrote and rewrote each of the chapters several times as I received feedback and as my understanding of how I wanted to communicate a topic improved.

Iterative development is an excellent way to improve the product as it is being developed. The biggest downside to iterative development is that in the presence of uncertainty it can be difficult up front to determine (plan) how many improvement passes will be necessary.

Incremental development is based on the age-old principle of "Build some of it before you build all of it." We avoid having one large, *big-bang-style* event at the end of development where all the pieces come together and the entire product is delivered. Instead, we break the product into smaller pieces so that we can build some of it, learn how each piece is to survive in the environment in which it must exist, adapt based on what we learn, and then build more of it. While writing this book, I wrote a chapter at a time and sent each chapter out for review as it was completed, rather than trying to receive feedback on the entire book at once. This gave me the opportunity to incorporate that feedback into future chapters, adjusting my tone, style, or delivery as needed. It also gave me the opportunity to learn incrementally and apply what I learned from earlier chapters to later chapters.

Incremental development gives us important information that allows us to adapt our development effort and to change how we proceed. The biggest drawback to incremental development is that by building in pieces, we risk missing the big picture (we see the trees but not the forest).

Scrum leverages the benefits of both iterative and incremental development, while negating the disadvantages of using them individually. Scrum does this by using both ideas in an adaptive series of timeboxed iterations called sprints (see Figure 3.4).

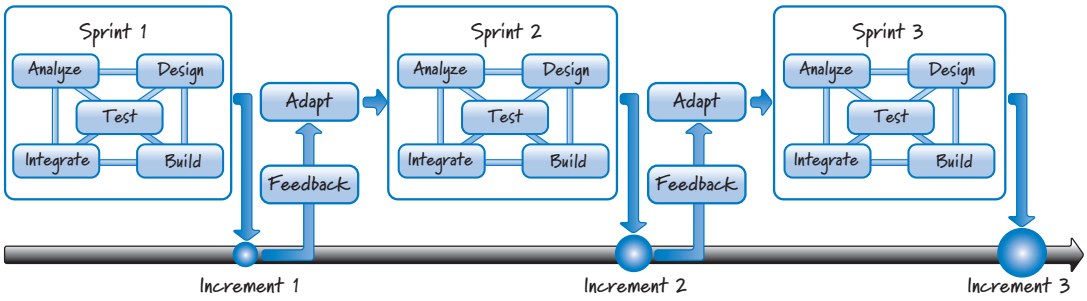

FIGURE 3.4 Scrum uses iterative and incremental development.

During each sprint we perform all of the activities necessary to create a working product increment (some of the product, not all of it). This is illustrated in Figure 3.4 by showing that some analysis, design, build, integration, and test work is completed in each sprint. This all-at-once approach has the benefit of quickly validating the assumptions that are made when developing product features. For example, we make some design decisions, create some code based on those decisions, and then test the design and code—all in the same sprint. By doing all of the related work within one sprint, we are able to quickly rework features, thus achieving the benefits of iterative development, without having to specifically plan for additional iterations.

A misuse of the sprint concept is to focus each sprint on just one type of work—for example, sprint 1 (analysis), sprint 2 (design), sprint 3 (coding), and sprint 4 (testing). Such an approach attempts to overlay Scrum with a waterfall-style work breakdown structure. I often refer to this misguided approach as **WaterScrum**, and I have heard others refer to it as **Scrummerfall**.

In Scrum, we don't work on a phase at a time; we work on a feature at a time. So, by the end of a sprint we have created a valuable product increment (some but not all of the product features). That increment includes or is integrated and tested with any previously developed features; otherwise, it is not considered done. For example, increment 2 in Figure 3.4 includes the features of increment 1. At the end of the sprint, we can get feedback on the newly completed features within the context of already completed features. This helps us view the product from more of a big-picture perspective than we might otherwise have.

We receive feedback on the sprint results, which allows us to adapt. We can choose different features to work on in the next sprint or alter the process we will use to build the next set of features. In some cases, we might learn that the increment, though it technically fits the bill, isn't as good as it could be. When that happens, we can schedule rework for a future sprint as part of our commitment to iterative development and continuous improvement. This helps overcome the issue of not knowing

up front exactly how many improvement passes we will need. Scrum does not require that we predetermine a set number of iterations. The continuous stream of feedback will guide us to do the appropriate and economically sensible number of iterations while developing the product incrementally.

Leverage Variability through Inspection, Adaptation, and Transparency

Plan-driven processes and Scrum are fundamentally different along several dimensions (see Table 3.1, based on dimensions suggested by Reinertsen 2009a).

A plan-driven, sequential development process assumes little or no output variability. It follows a well-defined set of steps and uses only small amounts of feedback late in the process. In contrast, Scrum embraces the fact that in product development, some level of variability is required in order to build something new. Scrum also assumes that the process necessary to create the product is complex and therefore would defy a complete up-front definition. Furthermore, it generates early and frequent feedback to ensure that the right product is built and that the product is built right.

At the heart of Scrum are the principles of **inspection**, **adaptation,** and **transparency** (referred to collectively by Schwaber and Beedle 2001 as **empirical process control**). In Scrum, we inspect and adapt not only what we are building but also how we are building it (see Figure 3.5).

To do this well, we rely on transparency: all of the information that is important to producing a product must be available to the people involved in creating the product. Transparency makes inspection possible, which is needed for adaptation. Transparency also allows everyone concerned to observe and understand what is happening. It leads to more communication and it establishes trust (both in the process and among team members).

TABLE 3.1 Comparison of Plan-Driven and Scrum Processes

Dimension	Plan-Driven	Scrum
Degree of process definition	Well-defined set of sequential steps	Complex process that would defy a complete up-front definition
Randomness of output	Little or no output variability	Expect variability because we are not trying to build the same thing over and over
Amount of feedback used	Little and late	Frequent and early

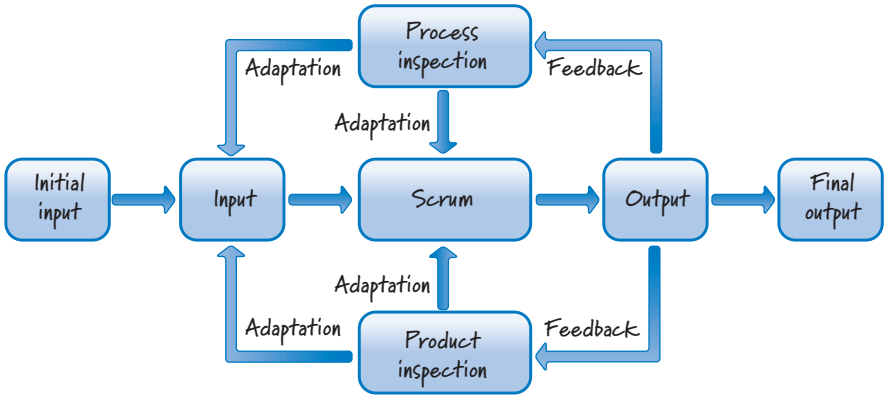

FIGURE 3.5 Scrum process model

Reduce All Forms of Uncertainty Simultaneously

Developing new products is a complex endeavor with a high degree of uncertainty. That uncertainty can be divided into two broad categories (Laufer 1996):

- **End uncertainty** (*what* uncertainty)—uncertainty surrounding the features of the final product
- **Means uncertainty** (*how* uncertainty)—uncertainty surrounding the process and technologies used to develop a product

In particular environments or with particular products there might also be **customer uncertainty** (*who* uncertainty). For example, start-up organizations (including large organizations that focus on novel products) may only have assumptions as to who the actual customers of their products will be. This uncertainty must be addressed or they might build brilliant products for the wrong markets.

Traditional, sequential development processes focus first on eliminating all end uncertainty by fully defining up front what is to be built, and only then addressing means uncertainty.

This simplistic, linear approach to uncertainty reduction is ill suited to the complex domain of product development, where our actions and the environment in which we operate mutually constrain one another. For example:

- We decide to build a feature (our action).
- We then show that feature to a customer, who, once he sees it, changes his mind about what he really wants, or realizes that he did not adequately convey the details of the feature (our action elicits a response from the environment).

- We make design changes based on the feedback (the environment's reaction influences us to take another unforeseen action).

In Scrum, we do not constrain ourselves by fully addressing one type of uncertainty before we address the next type. Instead, we take a more holistic approach and focus on simultaneously reducing all uncertainties (end, means, customer, and so on). Of course, at any point in time we might focus more on one type of uncertainty than another. Simultaneously addressing multiple types of uncertainty is facilitated by iterative and incremental development and guided by constant inspection, adaptation, and transparency. Such an approach allows us to opportunistically probe and explore our environment to identify and learn about the **unknown unknowns** (the things that we don't yet know that we don't know) as they emerge.

Prediction and Adaptation

When using Scrum, we are constantly balancing the desire for prediction with the need for adaptation. I describe five agile principles related to this topic:

- Keep options open.
- Accept that you can't get it right up front.
- Favor an adaptive, exploratory approach.
- Embrace change in an economically sensible way.
- Balance predictive up-front work with adaptive just-in-time work.

Keep Options Open

Plan-driven, sequential development requires that important decisions in areas like requirements or design be made, reviewed, and approved within their respective phases. Furthermore, these decisions must be made before we can transition to the next phase, even if those decisions are based on limited knowledge.

Scrum contends that we should never make a premature decision just because a generic process would dictate that now is the appointed time to make one. Instead, when using Scrum, we favor a strategy of keeping our options open. Often this principle is referred to as the **last responsible moment** (**LRM**) (Poppendieck and Poppendieck 2003), meaning that we delay commitment and do not make important and irreversible decisions until the last responsible moment. And when is that? When the cost of not making a decision becomes greater than the cost of making a decision (see Figure 3.6). At that moment, we make the decision.

To appreciate this principle, consider this. On the first day of a product development effort we have the least information about what we are doing. On each subsequent day of the development effort, we learn a little more. Why, then, would we ever choose to make all of the most critical, and perhaps irreversible, decisions on the first

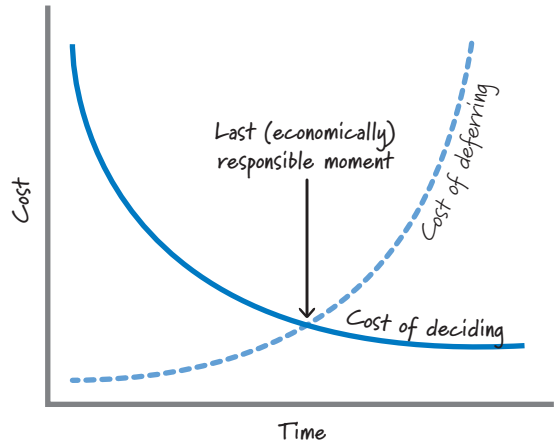

FIGURE 3.6 Make decisions at the last responsible moment.

day or very early on? Most of us would prefer to wait until we have more information so that we can make a more informed decision. When dealing with important or irreversible decisions, if we decide too early and are wrong, we will be on the exponential part of the cost-of-deciding curve in Figure 3.6. As we acquire a better understanding regarding the decision, the cost of deciding declines (the likelihood of making a bad decision declines because of increasing market or technical certainty). That's why we should wait until we have better information before committing to a decision.

Accept That You Can't Get It Right Up Front

Plan-driven processes not only mandate full requirements and a complete plan; they also assume that we can "get it right" up front. The reality is that it is very unlikely that we can get all of the requirements, or the detailed plans based on those requirements, correct up front. What's worse is that when the requirements do change, we have to modify the baseline requirements and plans to match the current reality (more about this in Chapter 5).

In Scrum, we acknowledge that we can't get all of the requirements or the plans right up front. In fact, we believe that trying to do so could be dangerous because we are likely missing important knowledge, leading to the creation of a large quantity of low-quality requirements (see Figure 3.7).

This figure illustrates that when using a plan-driven, sequential process, a large number of requirements are produced early on when we have the least cumulative knowledge about the product. This approach is risky, because there is an illusion that we have eliminated end uncertainty. It is also potentially very wasteful when our understanding improves or things change (as I will describe shortly).

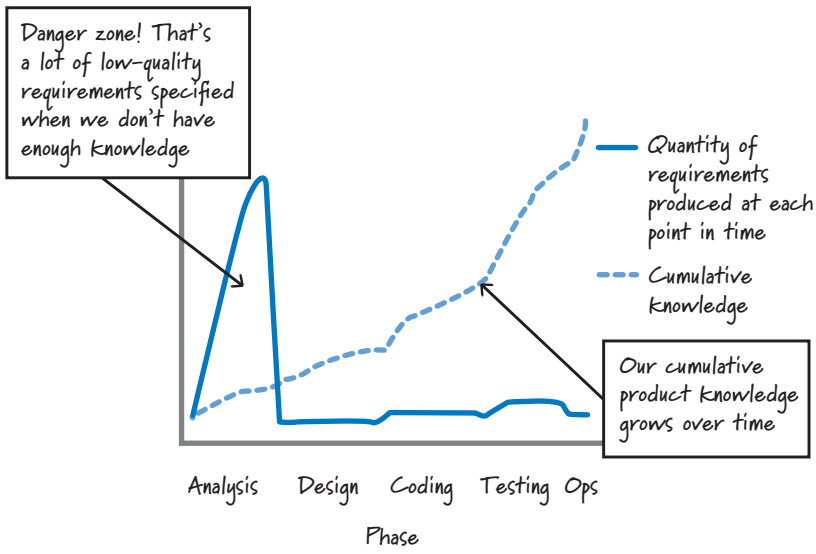

FIGURE 3.7 Plan-driven requirements acquisition relative to product knowledge

With Scrum, we still produce some requirements and plans up front, but just sufficiently, and with the assumption that we will fill in the details of those requirements and plans as we learn more about the product we are building. After all, even if we think we're 100% certain about what to build and how to organize up front the work to build it, we will learn where we are wrong as soon as we subject our early incremental deliverables to the environment in which they must exist. At that point all of the inconvenient realities of what is really needed will drive us to make changes.

Favor an Adaptive, Exploratory Approach

Plan-driven, sequential processes focus on using (or exploiting) what is currently known and predicting what isn't known. Scrum favors a more adaptive, trial-and-error approach based on appropriate use of exploration.

Exploration refers to times when we choose to gain knowledge by doing some activity, such as building a prototype, creating a proof of concept, performing a study, or conducting an experiment. In other words, when faced with uncertainty, we buy information by exploring.

Our tools and technologies significantly influence the cost of exploration. Historically software product development exploration has been expensive, a fact that favored a more predictive, try-to-get-it-right-up-front approach (see Figure 3.8).

As an example, in my freshman year at Georgia Tech (early 1980s), I (briefly) used punch cards—a tool that, like a typewriter, made you loathe to make any

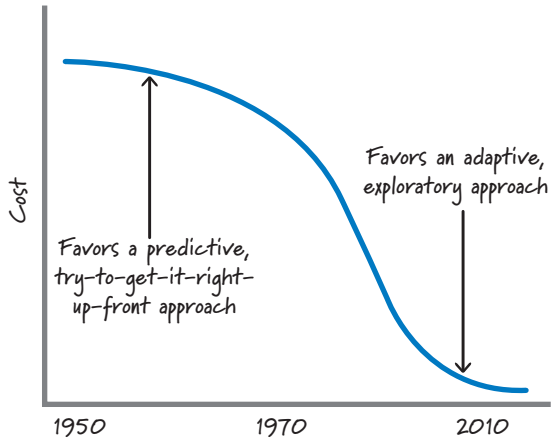

FIGURE 3.8 Historical cost of exploration

mistakes or modifications. It was hard to embrace the concept of "Let's quickly try that out and see what happens" when, with each potential solution, you had to painstakingly create punch cards, get in the queue for the mainframe, and wait up to 24 hours to get validation of your solution. Even the cost of a simple typo was at least a day in the schedule. A waterfall-style process that allowed for careful consideration of current knowledge and prediction in the presence of uncertainty in an attempt to arrive at a good solution just made economic sense.

Fortunately, tools and technologies have gotten better and the cost of exploring has come way down. There is no longer an economic disincentive to explore. In fact, nowadays, it's often cheaper to adapt to user feedback based on building something fast than it is to invest in trying to get everything right up front. Good thing, too, because the context (the surrounding technologies) in which our solutions must exist is getting increasingly more complex.

In Scrum, if we have enough knowledge to make an informed, reasonable step forward with our solution, we advance. However, when faced with uncertainty, rather than trying to predict it away, we use low-cost exploration to buy relevant information that we can then use to make an informed, reasonable step forward with our solution. The feedback from our action will help us determine if and when we need further exploration.

Embrace Change in an Economically Sensible Way

When using sequential development, change, as we have all learned, is substantially more expensive late than it is early on (see Figure 3.9, based on Boehm 1981).

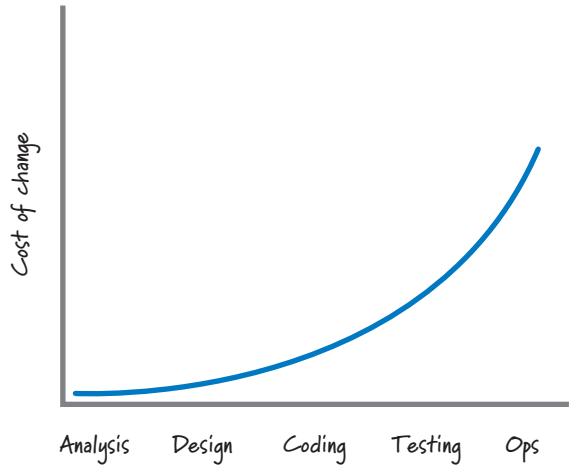

FIGURE 3.9 Significant late cost of change with sequential development

As an example, a change made during analysis might cost $1; that same change made late during testing might cost $1,000. Why is this so? If we make a mistake during analysis and find it during analysis, it is an inexpensive fix. If that same error is not found until design, we have to fix not only the incorrect requirement, but potentially parts of our design based on the wrong requirement. This compounding of the error continues through each subsequent phase, making what might have been a small error to correct during analysis into a much larger error to correct in testing or operations.

To avoid late changes, sequential processes seek to carefully control and minimize any changing requirements or designs by improving the accuracy of the predictions about what the system needs to do or how it is supposed to do it.

Unfortunately, being excessively predictive in early-activity phases often has the opposite effect. It not only fails to eliminate change; it actually contributes to deliveries that are late and over budget as well. Why this paradoxical truth? First, the desire to eliminate expensive change forces us to overinvest in each phase—doing more work than is necessary and practical. Second, we're forced to make decisions based on important assumptions early in the process, before we have validated these assumptions with feedback from our stakeholders based on our working assets. As a result, we produce a large inventory of work products based on these assumptions. Later, this inventory will likely have to be corrected or discarded as we validate (or invalidate) our assumptions, or change happens (for example, requirements emerge or evolve), as it always will. This fits the classic pattern of a self-fulfilling prophecy (see Figure 3.10).

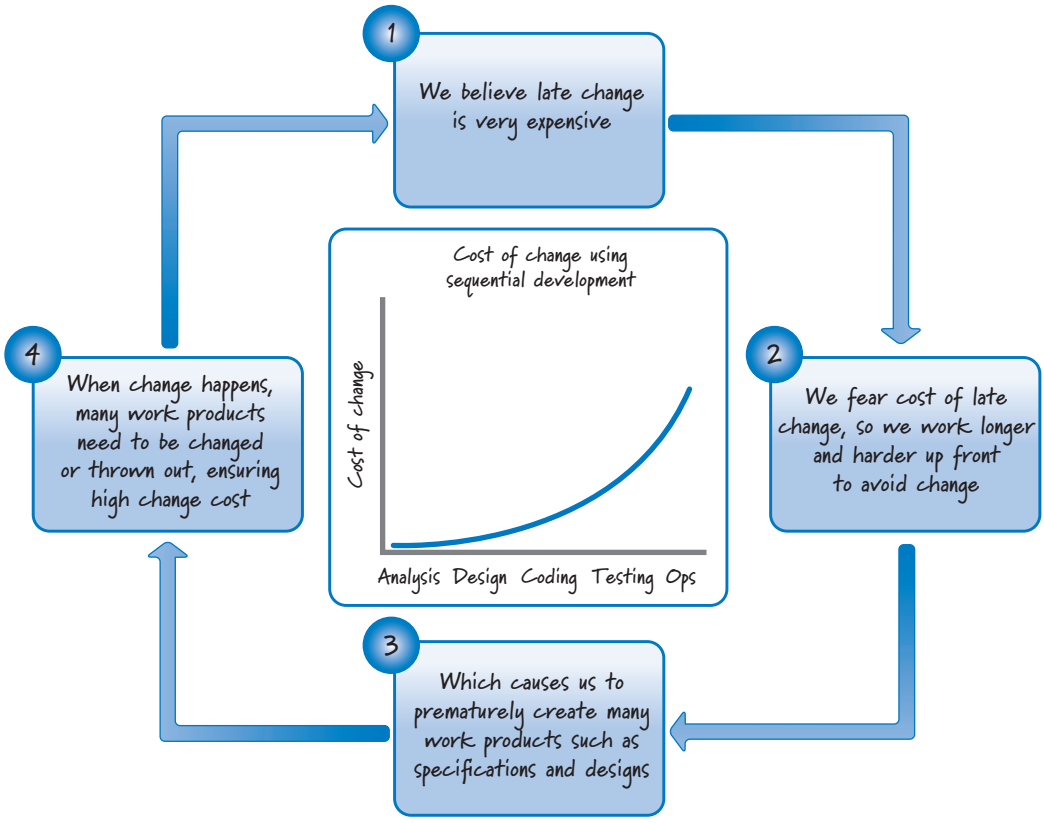

FIGURE 3.10 Self-fulfilling prophecy

In Scrum, we assume that change is the norm. We believe that we can't predict away the inherent uncertainty that exists during product development by working longer and harder up front. Thus, we must be prepared to embrace change. And when that change occurs, we want the economics to be more appealing than with traditional development, even when the change happens later in the product development effort.

Our goal, therefore, is to keep the cost-of-change curve flat for as long as possible—making it economically sensible to embrace even late change. Figure 3.11 illustrates this idea.

We can achieve that goal by managing the amount of work in process and the flow of that work so that the cost of change when using Scrum is less affected by time than it is with sequential projects.

Regardless of which product development approach we use, we want the following relationship to be true: a small change in requirements should yield a proportionally

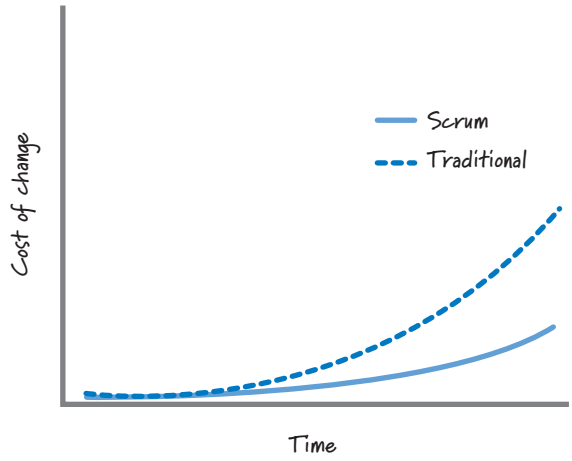

FIGURE 3.11 Flattening the cost-of-change curve

small change in implementation and therefore in cost (obviously we would expect a larger change to cost more). Another desirable property of this relationship is that we want it to be true regardless of *when* the change request is made.

With Scrum, we produce many work products (such as detailed requirements, designs, and test cases) in a just-in-time fashion, avoiding the creation of potentially unnecessary artifacts. As a result, when a change is made, there are typically far fewer artifacts or constraining decisions based on assumptions that might be discarded or reworked, thus keeping the cost more proportional to the size of the requested change.

Using sequential development, the early creation of artifacts and push for premature decision making ultimately mean that the cost of a change rises rapidly over time as inventory grows. This causes the inflection point (where the line begins to aggressively climb up) on the traditional curve in Figure 3.11 to occur early. When developing with Scrum, there does come a time when the cost of change will no longer be proportional to the size of the request, but this point in time (as illustrated by the inflection point on the Scrum curve in Figure 3.11) occurs later.

Balance Predictive Up-Front Work with Adaptive Just-in-Time Work

A fundamental belief of plan-driven development is that detailed up-front requirements and planning are critical and should be completed before moving on to later stages. In Scrum, we believe that up-front work should be helpful without being excessive.

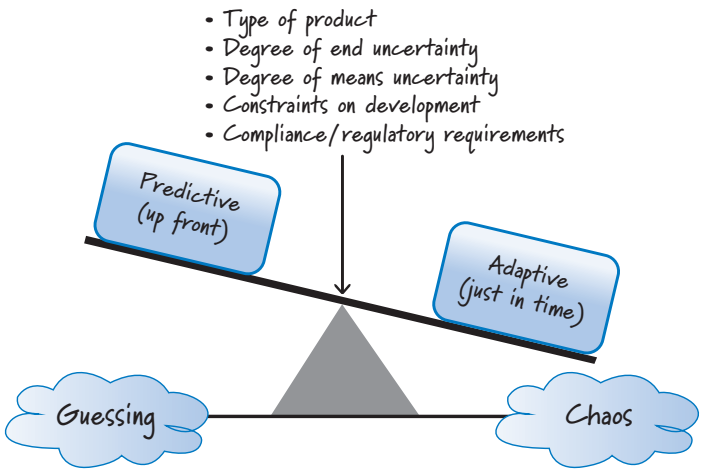

FIGURE 3.12 Balancing predictive and adaptive work

With Scrum, we acknowledge that it is not possible to get requirements and plans precisely right up front. Does that mean we should do no requirements or planning work up front? Of course not! Scrum is about finding balance—balance between predictive up-front work and adaptive just-in-time work (see Figure 3.12, adapted from a picture by Cohn 2009).

When developing a product, the balance point should be set in an economically sensible way to maximize the amount of ongoing adaptation based on fast feedback and minimize the amount of up-front prediction, while still meeting compliance, regulatory, and/or corporate objectives.

Exactly how that balance is achieved is driven in part by the type of product being built, the degree of uncertainty that exists in both what we want to build (end uncertainty) and how we want to build it (means uncertainty), and the constraints placed on the development. Being overly predictive would require us to make many assumptions in the presence of great uncertainty. Being overly adaptive could cause us to live in a state of constant change, making our work feel inefficient and chaotic. To rapidly develop innovative products we need to operate in a space where adaptability is counterbalanced by just enough prediction to keep us from sliding into chaos. The Scrum framework operates well at this balance point of order and chaos.

Validated Learning

When using Scrum, we organize the work to quickly create **validated learning** (a term proposed by Ries 2011). We acquire validated learning when we obtain knowledge

that confirms or refutes an assumption that we have made. I describe three agile principles related to this topic:

- Validate important assumptions fast.
- Leverage multiple concurrent learning loops.
- Organize workflow for fast feedback.

Validate Important Assumptions Fast

An **assumption** is a guess, or belief, that is assumed to be true, real, or certain even though we have no validated learning to know that it is true. Plan-driven development is much more tolerant of long-lived assumptions than Scrum. Using plan-driven development, we produce extensive up-front requirements and plans that likely embed many important assumptions, ones that won't be validated until a much later phase of development.

Assumptions represent a significant development **risk**. In Scrum, we try to minimize the number of important assumptions that exist at any time. We also don't want to let important assumptions exist without validation for very long. The combination of iterative and incremental development along with a focus on low-cost exploration can be used to validate assumptions fast. As a result, if we make a fundamentally bad assumption when using Scrum, we will likely discover our mistake quickly and have a chance to recover from it. In plan-driven, sequential development, the same bad assumption if validated late might cause a substantial or total failure of the development effort.

Leverage Multiple Concurrent Learning Loops

There is learning that occurs when using sequential development. However, an important form of learning happens only after features have been built, integrated, and tested, which means considerable learning occurs toward the end of the effort. Late learning provides reduced benefits because there may be insufficient time to leverage the learning or the cost to leverage it might be too high.

In Scrum, we understand that constant learning is a key to our success. When using Scrum, we identify and exploit feedback loops to increase learning. A recurring pattern in this style of product development is to make an assumption (or set a goal), build something (perform some activities), get feedback on what we built, and then use that feedback to inspect what we did relative to what we assumed. We then make adaptations to the product, process, and/or our beliefs based on what we learned (see Figure 3.13).

Scrum leverages several predefined **learning loops**. For example, the daily scrum is a daily loop and the sprint review is an iteration-level loop. I will describe these and others in subsequent chapters.

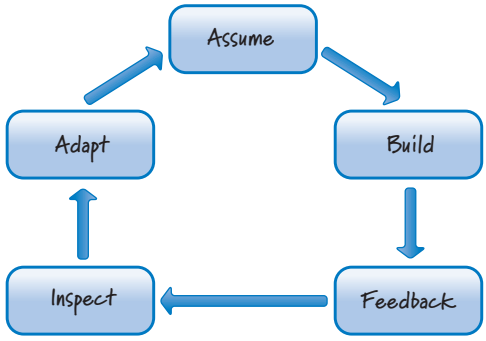

FIGURE 3.13 Learning loop pattern

The Scrum framework is also flexible enough to embrace many other learning loops. For example, although not specified by Scrum, technical practice feedback loops, such as pair programming (feedback in seconds) and test-driven development (feedback in minutes), are frequently used with Scrum development.

Organize Workflow for Fast Feedback

Being tolerant of long-lived assumptions also makes plan-driven processes tolerant of late learning, so **fast feedback** is not a focus. With Scrum, we strive for fast feedback, because it is critical for helping truncate wrong paths sooner and is vital for quickly uncovering and exploiting time-sensitive, emergent opportunities.

In a plan-driven development effort, every activity is planned to occur at an appointed time based on the well-defined phase sequence. This approach assumes that earlier activities can be completed without the feedback generated by later activities. As a result, there might be a long period of time between doing something and getting feedback on what we did (hence closing the learning loop).

Let's use component **integration** and testing as an example. Say we are developing three components in parallel. At some time these components have to be integrated and tested before we have a shippable product. Until we try to do the integration, we really don't know whether we have developed the components correctly. Attempting the integration will provide critical feedback on the component development work.

Using sequential development, integration and testing wouldn't happen until the predetermined downstream phase, where many or all components would be integrated. Unfortunately, the idea that we can develop a bunch of components in parallel and then later, in an integration phase, smoothly bring them together into a cohesive whole is unlikely to work out. In fact, even with well-conceived interfaces defined before we develop the components, it's likely that something will go wrong when we integrate them (see Figure 3.14).

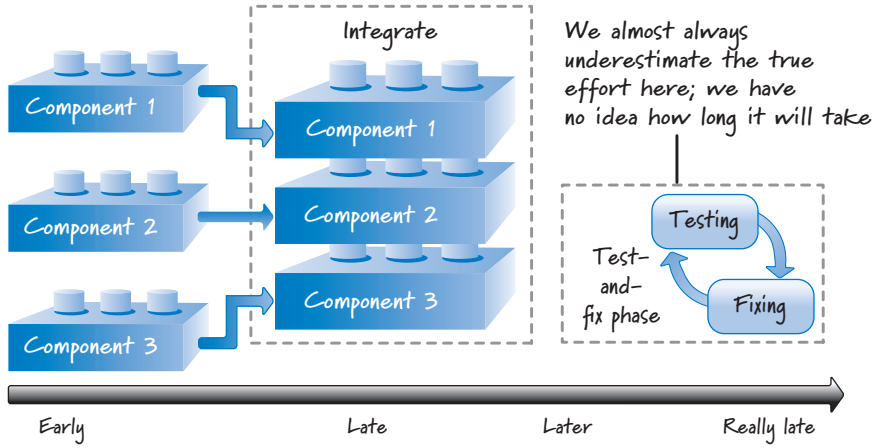

FIGURE 3.14 Component integration

Feedback-generating activities that occur a long time after development have unfortunate side effects, such as turning integration into a large test-and-fix phase, because components developed disjointedly from each other frequently don't integrate smoothly. How long it will take and how much it will cost to fix the problem can only be guessed at this point.

In Scrum, we organize the flow of work to move through the learning loop in Figure 3.13 and get to feedback as quickly as possible. In doing so, we ensure that feedback-generating activities occur in close time proximity to the original work. Fast feedback provides superior economic benefits because errors compound when we delay feedback, resulting in exponentially larger failures.

Let's look again at our component integration example. When we designed the components, we made important assumptions about how they would integrate. Based on those assumptions, we proceeded down a design path. We do not, at this point, know whether the selected design path is right or wrong. It's just our best guess.

Once we choose a path, however, we then make many other decisions that are based on that choice. The longer we wait to validate the original design assumption, the greater the number of dependent decisions. If we later determine (via feedback during the integration phase) that the original assumption was wrong, we'll have a large, compounded mess on our hands. Not only will we have many bad decisions that have to be reworked; we'll also have to do it after a great deal of time has passed. Because people's memories will have faded, they will spend time getting back up to speed on the work they did earlier.

When we factor in the total cost of reworking potentially bad dependent decisions, and the cost of the delay to the product, the economic benefits of fast feedback are very compelling. Fast feedback closes the learning loop quickly, allowing us to truncate bad development paths before they can cause serious economic damage.

Work in Process (WIP)

Work in process (or **WIP**) refers to the work that has been started but not yet finished. During product development WIP must be recognized and properly managed. I describe four agile principles related to this topic:

- Use economically sensible batch sizes.
- Recognize inventory and manage it for good flow.
- Focus on idle work, not idle workers.
- Consider cost of delay.

Use Economically Sensible Batch Sizes

Another core belief underlying plan-driven, sequential development processes is that it is preferable to batch up all of one type of work and perform it in a single phase. I refer to this as the **all-before-any** approach, where we complete all (or substantially all) of one activity before starting the next. Let's say we create all of the requirements during the analysis phase. Next, we move the batch of requirements into the design phase. Because we generated the complete set of requirements, our **batch size** in this example is 100%.

The all-before-any approach is, in part, a consequence of believing that the old manufacturing principle of economies of scale applies to product development. This principle states that the cost of producing a unit will go down as we increase the number of units (the batch size) that are produced. So, the sequential development belief is that larger batches in product development will also realize economies of scale.

In Scrum, we accept that although economies-of-scale thinking has been a bedrock principle in manufacturing, applying it dogmatically to product development will cause significant economic harm.

As counterintuitive as it might sound, working in smaller batches during product development has many benefits. Reinertsen discusses batch-size issues in depth, and Table 3.2 includes a subset of the small-batch-size benefits that he describes (Reinertsen 2009b).

If small batches are better than large batches, shouldn't we just use a batch size of one, meaning that we work on only one requirement at a time and flow it through all of the activities until it is done and ready for a customer? Some people refer to this as **single-piece flow**. As I will show in later chapters, a batch size of one might be appropriate in some cases, but assuming that "one" is the goal might suboptimize the flow and our overall economics.

TABLE 3.2 Reinertsen Benefits of Small Batch Sizes

Benefit	Description
Reduced cycle time	Smaller batches yield smaller amounts of work waiting to be processed, which in turn means less time waiting for the work to get done. So, we get things done faster.
Reduced flow variability	Think of a restaurant where small parties come and go (they flow nicely through the restaurant). Now imagine a large tour bus (large batch) unloading and the effect that it has on the flow in the restaurant.
Accelerated feedback	Small batches accelerate fast feedback, making the consequences of a mistake smaller.
Reduced risk	Small batches represent less inventory that is subject to change. Smaller batches are also less likely to fail (there is a greater risk that a failure will occur with ten pieces of work than with five).
Reduced overhead	There is overhead in managing large batches—for example, maintaining a list of 3,000 work items requires more effort than a list of 30.
Increased motivation and urgency	Small batches provide focus and a sense of responsibility. It is much easier to understand the effect of delays and failure when dealing with small versus large batches.
Reduced cost and schedule growth	When we're wrong on big batches, we are wrong in a big way with respect to cost and schedule. When we do things on a small scale, we won't be wrong by much.

Recognize Inventory and Manage It for Good Flow

Throughout this chapter, I have been reminding you that manufacturing and product development are not the same thing and thus should be approached differently. There is, however, one lesson that manufacturing has learned that we should apply to product development and yet often do not. That lesson has to do with the high cost of inventory, also known as work in process or WIP. The lean product development community has known for many years of the importance of WIP (Poppendieck and Poppendieck 2003; Reinertsen 2009b), and Scrum teams embrace this concept.

Manufacturers are acutely aware of their inventories and the financial implications of those inventories. How can they not be? Inventory quickly starts to pile up on the floor, waiting to be processed. Not only is factory inventory physically visible; it is also financially visible. Ask the CFO of a manufacturing company how much

inventory (or WIP) he has in the factory or how much it has changed in the past month and he can give a definitive answer.

No competent manufacturer sits on a large quantity of inventory. Parts that are sitting on the factory floor waiting to be put into finished goods are depreciating on the financial books. Worse yet, what happens if we purchase a truckload of parts, and then change the design of the product? What do we do with all of those parts? Maybe we rework the parts so that they fit into the new design. Or worse, maybe we discard the parts because they can no longer be used. Or, to avoid incurring waste on the parts we already purchased, are we going to not change our design (even though doing so would be the correct design choice) so we can use those parts—at the risk of producing a less satisfying product?

It's obvious that if we sit on a lot of inventory and then something changes, we experience one or more forms of significant waste. To minimize risks, competent manufacturers manage inventory in an economically sensible way—they keep some inventory on hand but use a healthy dose of just-in-time inventory management.

Product development organizations, generally speaking, are not nearly as cognizant of their work in process. Part of the problem stems from the fact that in product development we deal with knowledge assets that aren't physically visible in the same way as parts on the factory floor. Knowledge assets are far less intrusive, such as code on a disk, a document in a file cabinet, or a visual board on the wall.

Inventory in product development is also typically not financially visible. Ask a CFO of a product development organization how much inventory exists in the product development organization and he will likely give you a puzzled look and say, "None." While the financial team is tracking other measures of a product development effort, it won't likely be tracking product development inventory of this type.

Unfortunately, inventory (WIP) is a critical variable to be managed during product development, and the traditional approaches to product development don't focus on managing it. As I mentioned in the discussion of batch sizes, by setting the batch size to be quite large (frequently 100%), traditional development actually favors the creation of large amounts of inventory. An important consequence of having a lot of WIP in product development is that it significantly affects the cost-of-change curve I described earlier (see Figure 3.9).

Although we need some requirements if we are going to start development, we don't need to have *all* of the requirements. If we have too many requirements, we will likely experience inventory waste when requirements change. On the other hand, if we don't have enough requirements inventory, we will disrupt the fast flow of work, which is also a form of waste. In Scrum, our goal is to find the proper balance between just enough inventory and too much inventory.

It is important to realize that requirements are just one form of inventory that exists in product development. There are many different places and times during product development where we have WIP. We need to proactively identify and manage those as well.

Focus on Idle Work, Not Idle Workers

In Scrum, we believe that idle work is far more wasteful and economically damaging than idle workers. **Idle work** is work that we want to do (such as building or testing something) but can't do because something is preventing us. Perhaps we are blocked waiting on another team to do something, and until that team completes its work, we can't do ours. Or maybe we just have so much work to do that it can't all be done at once. In this case, some of the work sits idle until we become available to work on it. **Idle workers**, on the other hand, are people who have available **capacity** to do more work because they are not currently 100% utilized.

Many product development organizations focus more on eliminating the waste of idle workers than on the waste of idle work. For example, in traditional thinking, if I hire you to be a tester, I expect you to spend 100% of your time testing. If you spend less than 100% of your time testing, I incur waste (you're idle when you could be testing). To avoid this problem, I will find you more testing work to do—perhaps by assigning you to multiple projects—to get your utilization up to 100%.

Unfortunately, this approach reduces one form of waste (idle-worker waste) while simultaneously increasing another form of waste (idle-work waste). And, most of the time, the cost of the idle work is far greater than the cost of an idle worker. Let's explore why this is true.

To illustrate the issue, let's apply the keep-workers-100%-busy strategy to the 4 × 100-meter relay race at the Olympics. Based on the keep-them-busy strategy, this race seems highly inefficient. I pay people to run and they seem to be running only one-quarter of the time. The rest of the time they are just standing around. Well, that's not right! I pay them 100% salary so I want them to run 100% of the time. How about if when they're not carrying the baton, they just run up and down the stands or perhaps run another race on an adjacent track? That way they will be utilized 100% at running.

Of course, we all know that you don't win the relay gold medal by keeping the runners 100% busy. You win the gold medal by getting the baton across the finish line first. So, the important takeaway is "Watch the baton, not the runners" (Larman and Vodde 2009). In the context of product development, the baton sitting on the ground equates to work that is ready to be performed but is blocked waiting for necessary resources. You don't win the race (deliver products) when the baton is on the ground. (I really like the baton and runner analogy because it nicely illustrates that we should watch the work and not the workers. However, like any analogy, it has its limits. In this case, the relay-race approach to handing off work is precisely one aspect of traditional, sequential product development that we would like to avoid!)

Also, everyone knows the consequences of keeping a resource 100% busy. If we borrow a graph from queuing theory, we can see the obvious damage caused when striving for 100% utilization (see Figure 3.15).

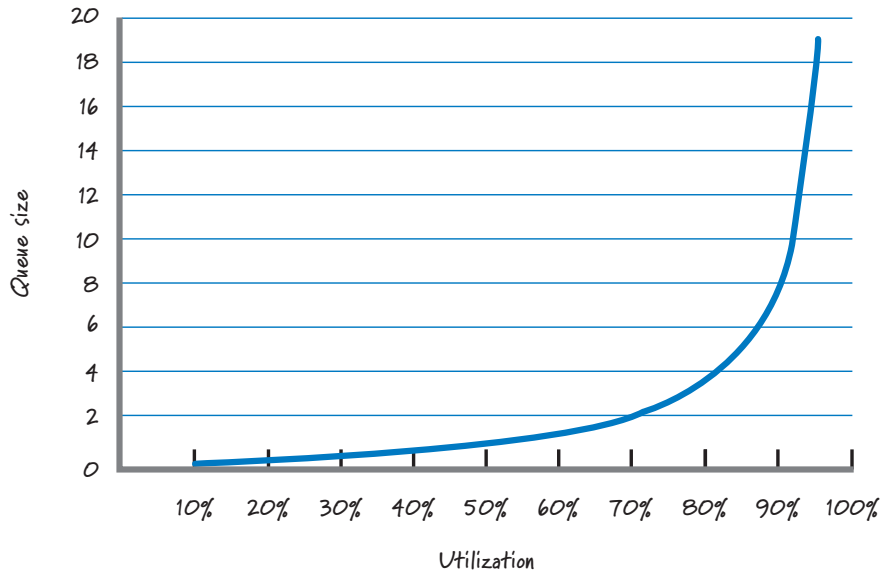

FIGURE 3.15 How utilization affects queue size (delay)

Anyone who owns a computer understands this graph. What happens if you run your computer at 100% (full processor and memory utilization)? It starts to thrash and every job on the computer slows down. In other words, the computer is working on more things and actually gets less productive work completed. Once you get into that state, it is very difficult to get out of it (you probably have to start killing jobs or reboot the machine). Your computer would be much more efficient if you ran it at closer to 80% utilization. In Figure 3.15, **queue** size equates to delay and delay to the baton sitting on the ground.

The idle work (delayed work) grows exponentially once we get into the high levels of utilization. And that idle work can be very expensive, frequently many times more expensive than the cost of idle workers (see the next section on cost of delay for an example). So, in Scrum, we are acutely aware that finding the bottlenecks in the flow of work and focusing our efforts on eliminating them is a far more economically sensible activity than trying to keep everyone 100% busy.

Consider Cost of Delay

Cost of delay is the financial cost associated with delaying work or delaying achievement of a milestone. Figure 3.15 illustrates that as capacity utilization increases, queue size and delay also increase. Therefore, by reducing the waste of idle workers (by increasing their utilization), we simultaneously increase the waste associated

with idle work (work sitting in queues waiting to be serviced). Using cost of delay, we can calculate which waste is more economically damaging.

Unfortunately, 85% of organizations don't quantify cost of delay (Reinertsen 2009b). Combine that with the fact that most development organizations don't realize they have accumulated work (inventory) sitting in queues, and it is easy to see why their default behavior is to focus on eliminating the visible waste of idle workers.

Here is a simple example to illustrate why the cost of idle work is typically much greater than the cost of idle workers. Consider this question: Should we assign a documenter to the team on the first day of development or at the end of development? Table 3.3 illustrates a comparison of these two options (there are other options we could use).

Assume that we assign the documenter full-time for 12 months to work on this product, even if he is not needed 100% of the time. Doing so costs an incremental $75K (think of this as idle worker waste) above what it would cost if we brought him on for two months at the end once the product reaches the state of "all but documented."

If we assign the documenter to do all of the documentation at the end, we will need him full-time for only two months, but we will also delay the delivery of the

TABLE 3.3 Example Cost-of-Delay Calculation

Parameter	Value
Duration with full-time documenter	12 months
Duration with documenter assigned at the end (when we reach the state of "all but documented")	14 months
Cycle-time cost for doing documentation at the end	2 months
Cost of delay, per month	$250K
Total cost of delay	**$500K**
Annual fully burdened cost of documenter	$90K
Monthly fully burdened cost of documenter	$7.5K
Cost for full-time documenter	$90K
Cost for documenter if assigned at end	$15K
Incremental cost for full-time documenter	**$75K**

product by the same two months. If we delay shipping the product by two months, the calculated cost of delay in terms of lifecycle profits is $500K (**lifecycle profits** are the total profit potential of a product over its lifetime; in this example, that potential decreases by $500K).

In this example, the cost of the idle worker is $75K and the cost of the idle work is $500K. If we focus on optimizing the utilization of the documenter, we will substantially suboptimize the economics of the overall product. During product development we are presented with these types of trade-offs on a continuous basis; cost of delay will be one of the most important variables to consider when making economically sensible decisions.

Progress

When using Scrum, we measure progress by what we have delivered and validated, not by how we are proceeding according to the predefined plan or how far we are into a particular phase or stage of development. I describe three agile principles related to this topic:

- Adapt to real-time information and replan.
- Measure progress by validating working assets.
- Focus on value-centric delivery.

Adapt to Real-Time Information and Replan

In a plan-driven, sequential process, the plan is the authoritative source on how and when work should occur. As such, conformance to the plan is expected. In contrast, in Scrum we believe that unbridled faith in the plan will frequently blind us to the fact that the plan might be wrong.

On a Scrum development effort our goal is not to conform to some plan, some up-front prediction of how we thought things might go. Instead, our goal is to rapidly replan and adapt to the stream of economically important information that is continuously arriving during the development effort.

Measure Progress by Validating Working Assets

Progress during a sequential, plan-driven development effort is demonstrated by completing a phase and being permitted to enter the next phase. As a result, if each phase starts and completes as expected, the product development effort might seem to be progressing quite well. Yet in the end, the product we created in full accordance with the plan might deliver far less customer value than anticipated. Can we really claim success if we finish on time and on budget and yet fail to meet customer expectations?

With Scrum, we measure progress by building working, validated assets that deliver value and that can be used to validate important assumptions. This gives us the feedback to know what the right next step is. In Scrum, it's not about how much work we start; it's all about what customer-valuable work we finish.

Focus on Value-Centric Delivery

Plan-driven, sequential development focuses on diligently following the process. By its very structure, the integration and delivery of features during sequential development happen at the end of the effort (see Figure 3.16). With this approach there is a risk that we will run out of resources (time or money) before we deliver all of the important value to our customers.

A related belief of traditional development is that the planning and document artifacts that get produced en route to delivering features are themselves valuable. If these artifacts are indeed valuable, most of the time they are valuable only to the downstream process and not the customers. And, if they are valuable to the customer, that value accrues only if a desirable product is ultimately delivered to the customer. Until that happens, these artifacts provide no direct customer value.

Scrum, on the other hand, is a customer-value-centric form of development. It is based on a prioritized, incremental model of delivery in which the highest-value features are continuously built and delivered in the next iteration. As a result, customers get a continuous flow of high-value features sooner.

In Scrum, value is generated by delivering working assets to customers, by validating important assumptions, or by acquiring valuable knowledge. In Scrum, we believe that the intermediate artifacts provide no perceived customer value and are merely a means to an end if they themselves cannot be used to generate important feedback or acquire important knowledge.

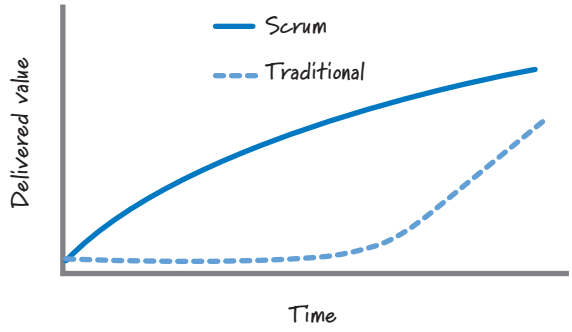

FIGURE 3.16 Deliver high-value features sooner.

Performance

There are specific performance-related characteristics we expect when using Scrum. I describe three agile principles related to this topic:

- Go fast but never hurry.
- Build in quality.
- Employ minimally sufficient ceremony.

Go Fast but Never Hurry

Plan-driven development believes that if we follow the plan and do things right the first time, we'll avoid costly and time-consuming rework. Moving from step to step quickly is of course desirable, but it isn't a principal goal.

In Scrum, one core goal is to be nimble, adaptable, and speedy. By going fast, we deliver fast, we get feedback fast, and we get value into the hands of our customers sooner. Learning and reacting quickly allow us to generate revenue and/or reduce costs sooner.

Do not, however, mistake going fast for being hurried. In Scrum, time is of the essence, but we don't rush to get things done. Doing so would likely violate the Scrum principle of **sustainable pace**—people should be able to work at a pace that they can continue for an extended period of time. In addition, hurrying will likely come at the expense of quality.

An example might help clarify the difference between fast and hurried. I study Muay Thai (Thai kickboxing). As is true of most martial arts, Muay Thai performance is enhanced with speed. Being able to swiftly and accurately perform katas or sparring enhances the pleasure of the sport and the outcome. However, hurrying through the movements with the intent of getting done substantially reduces their effectiveness and could cause serious bodily harm during sparring. When performing Muay Thai, you move swiftly, nimbly, and deliberately while quickly adapting to the situation. In other words, you need to be fast, but never hurried.

Build In Quality

During plan-driven development, the belief is that through careful, sequential performance of work we get a high-quality product. However, we can't actually verify this quality until we do the testing of the integrated product, which occurs during a late phase of the process. If testing should indicate that the quality is lacking, we then must enter the costly test-and-fix phase in an attempt to test quality in. Also, because a different team frequently works on each phase, the testing team is often viewed as owning the quality of the result.

In Scrum, quality isn't something a testing team "tests in" at the end; it is something that a cross-functional Scrum team owns and continuously builds in and

verifies every sprint. Each increment of value that is created is completed to a high level of confidence and has the potential to be put into production or shipped to customers (see Chapter 4 for a deeper discussion of the definition of done). As a result, the need for any significant late testing to tack on quality is substantially reduced.

Employ Minimally Sufficient Ceremony

Plan-driven processes tend to be high-**ceremony**, document-centric, process-heavy approaches. A side effect of Scrum's being value-centric is that very little emphasis is put on process-centric ceremonies. I don't mean to imply that all ceremony is bad. For example, a "ceremony" of going to the pub to socialize and bond every Friday after work would be a good ceremony. I am referring to ceremony that is **unnecessary formality**. Some might call it "process for the sake of process." Such ceremony has a cost but adds little or no value (in other words, it's a type of waste).

Example ceremonies that might be unnecessary formality include the following:

- A three-day, heavyweight process is required for approving and migrating code from the development environment to the QA environment before we are allowed to start testing.
- All anomalies have to be logged into a software tool so that they can be tracked and reported, even if I could just tap on the shoulder of the person sitting next to me and say, "Hey, this isn't working; could you fix it?" and have him make a fix so I can continue my work.
- I write a document because now is the prescribed time to write that document, even though nobody can say why that document is necessary or valuable.

In Scrum, our goal is to eliminate unnecessary formality. Therefore, we set the ceremonial bar at a low level, one that is minimally sufficient (some call it barely sufficient) or good enough. Of course, what constitutes minimally sufficient or good enough can differ from organization to organization. If we're building a new social media website, our need for ceremony might be exceptionally low. On the other hand, if we're building a pacemaker and are subject to numerous governmental regulations that require specific types of ceremonies, the minimally sufficient bar will be set higher (see Figure 3.17).

Frequently the Scrum focus on minimally sufficient ceremony is misinterpreted to mean things like "Scrum is anti-documentation." Scrum isn't anti-documentation. Rather, when using Scrum, we adopt an economic perspective and carefully review which documents we create. If we write a document that is shelfware and adds no value, we have wasted our time and money creating a dead document. However, not all documents are dead. For example, we will likely write a document if

- It is a deliverable as part of the product (for example, installation instructions, user's guide, and so on)

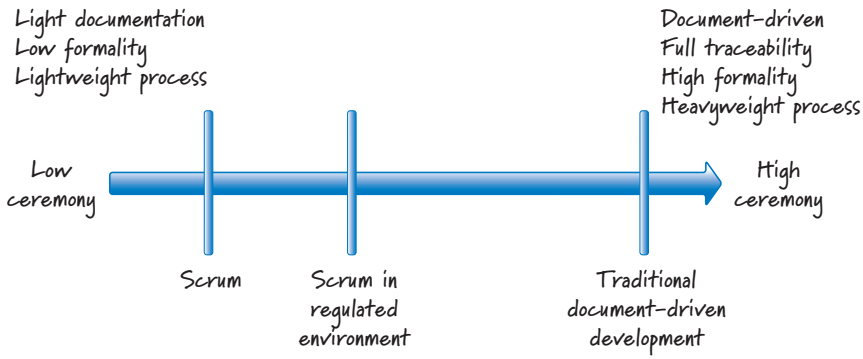

FIGURE 3.17 Ceremony scale

- Our goal is to capture an important discussion, decision, or agreement so that in the future we will have a clear recollection of what was discussed, decided, or agreed to
- It is the high-value way of helping new team members come up to speed quickly
- There is a regulatory requirement that a certain document be written (a cost of doing business in a regulated industry)

What we are trying to avoid is work that adds no short-term or long-term economic value. In Scrum, we believe that time and money are better spent delivering customer value.

Closing

In this chapter I focused on describing core agile principles—the fundamental beliefs that drive how we develop with Scrum. In doing so, I compared how these beliefs are different from the beliefs that underlie textbook, traditional, plan-driven, sequential development (which are summarized in Table 3.4).

My goal of making this comparison is not to convince you that waterfall is bad and that Scrum is good. Instead, my goal is to illustrate that the underlying beliefs of waterfall make it more appropriate to a different class of problem than Scrum. You can evaluate for yourself what type of problems your organization addresses and therefore which is the more appropriate tool to use. The subsequent chapters of this book will provide a detailed description of how these principles reinforce one another, providing a powerful approach to product development.

TABLE 3.4 Comparison Summary of Plan-Driven and Agile Principles

Topic	Plan-Driven Principle	Agile Principle
Similarity between development and manufacturing	Both follow a defined process.	Development isn't manufacturing; development creates the recipe for the product.
Process structure	Development is phase-based and sequential.	Development should be iterative and incremental.
Degree of process and product variability	Try to eliminate process and product variability.	Leverage variability through inspection, adaptation, and transparency.
Uncertainty management	Eliminate end uncertainty first, and then means uncertainty.	Reduce uncertainties simultaneously.
Decision making	Make each decision in its proper phase.	Keep options open.
Getting it right the first time	Assumes we have all of the correct information up front to create the requirements and plans.	We can't get it right up front.
Exploration versus exploitation	Exploit what is currently known and predict what isn't known.	Favor an adaptive, exploratory approach.
Change/emergence	Change is disruptive to plans and expensive, so it should be avoided.	Embrace change in an economically sensible way.
Predictive versus adaptive	The process is highly predictive.	Balance predictive up-front work with adaptive just-in-time work.
Assumptions (unvalidated knowledge)	The process is tolerant of long-lived assumptions.	Validate important assumptions fast.
Feedback	Critical learning occurs on one major analyze-design-code-test loop.	Leverage multiple concurrent learning loops.
Fast feedback	The process is tolerant of late learning.	Organize workflow for fast feedback.

continues

TABLE 3.4 Comparison Summary of Plan-Driven and Agile Principles (*Continued*)

Topic	Plan-Driven Principle	Agile Principle
Batch size (how much work is completed before the next activity can start)	Batches are large, frequently 100%—all before any. Economies of scale should apply.	Use smaller, economically sensible batch sizes.
Inventory/work in process (WIP)	Inventory isn't part of the belief system so is not a focus.	Recognize inventory and manage it to achieve good flow.
People versus work waste	Allocate people to achieve high levels of utilization.	Focus on idle work, not idle workers.
Cost of delay	Cost of delay is rarely considered.	Always consider cost of delay.
Conformance to plan	Conformance is considered a primary means of achieving a good result.	Adapt and replan rather than conform to a plan.
Progress	Demonstrate progress by progressing through stages or phases.	Measure progress by validating working assets.
Centricity	Process-centric—follow the process.	Value-centric—deliver the value.
Speed	Follow the process; do things right the first time and go fast.	Go fast but never hurry.
When we get high quality	Quality comes at the end, after an extensive test-and-fix phase.	Build quality in from the beginning.
Formality (ceremony)	Formality (well-defined procedures and checkpoints) is important to effective execution.	Employ minimally sufficient ceremony.